Exception Handling Patterns for Processes

Barbara Staudt Lerner

Department of Computer Science
Mount Holyoke College, South Hadley, MA 01075, USA
blerner@mtholyoke.edu

Stefan Christov, Alexander Wise, Leon J. Osterweil

Laboratory for Advanced Software Engineering Research (LASER)
University of Massachusetts at Amherst, Amherst, MA 01003
{christov, wise, ljo}@cs.umass.edu

Abstract. Exception handling patterns can raise the abstraction level of processes, facilitating their writing and understanding. In this paper, we identify several useful, general purpose exception handling patterns and demonstrate their applicability in software development process definition. We present these patterns using the Little-JIL language, which incorporates an exception handling construct that allows corrective actions to be taken in the case of exceptions, and flexible ways of defining how the process should continue after handling the exception.

Keywords. Exception handling, pattern, process language

1 Introduction

A process describes the activities and interactions of multiple agents working together to complete a task. Processes have been used for many different types of applications, such as software engineering, handling of insurance claims, describing medical procedures, conflict resolution and many others.

As in traditional programming, exceptions in processes represent situations that may be expected to occur occasionally but are not considered to be the normal situation. In traditional programming languages, it has become standard to provide exception handling mechanisms that allow a programmer to cleanly separate exception handling from normal processing. This makes the normal case far easier to understand and reduces the risk that an error will be missed because a status value returned by a function was not checked to uncover an error when it should have been.

As processes typically involve coordination of multiple people, there are many opportunities for problems to arise. People may be unavailable when they are needed,

the actions they take may not always reach the desired goals, deadlines might not be met, or a lack of resources may confound a group's ability to do their work. In each of these cases, an exceptional condition has arisen and appropriate action should be taken. Separating specification of these actions from specification of the actions that comprise the "normal" situation seems important in assuring the clarity of the process specification. Capturing the exceptions that might be anticipated within the process definition can also facilitate the planning and analysis of a process. Exception handling support within a process language facilitates the desired clear separation of exceptional behaviors from more normal behaviors.

Through our experience in defining processes in a variety of domains, we have realized that certain behaviors that recur frequently comprise specifiable patterns. In this paper, we focus on those patterns that demonstrate particularly effective use of exception handling. As with the burgeoning field of design patterns, we have found that by thinking in terms of patterns we are able to raise the level of abstraction associated with processes, making it easier both to create and to understand processes.

The exception handling patterns that we have observed fall into several broad categories of use:

- Presenting alternative means to perform the same activity.
- Inserting additional activities before returning to normal processing.
- Aborting the current processing.

In the remainder of this paper, we first describe the exception handling mechanism of Little-JIL and then discuss the exception handling patterns that we have found, using Little-JIL to elucidate the discussion and present examples.

2 Little-JIL

Little-JIL [1] is a hierarchically-scoped process language with a graphical syntax. The basic unit of Little-JIL processes is the step represented by the icon shown in Figure 1. Each step declares a scope and includes an interface, represented by the circle at the top of the figure, which specifies the information required and produced by the step. Pre- and post-requisites, represented by the triangles to the left and right sides of the step name, may be used to specify processes that are responsible for checking that the step can be performed, and that the step was performed correctly. Additionally, a step may be decomposed in two ways: substeps and exception handlers, discussed below.

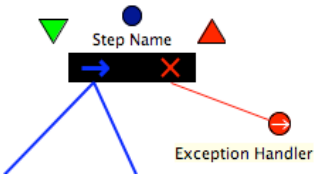
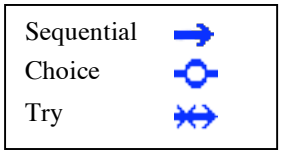


Figure 1: Little-JIL Syntax

2.1 Substeps

Little-JIL processes are constructed of steps decomposed into substeps. Substeps are connected to the icon on the left side of the black bar, via edges, which are annotated

with information regarding the binding of data between the steps. Each step specifies the execution order of its substeps using a sequencing icon, which appears in the black bar above the point where the substep edges are attached. There are three different sequencing icons used in this paper: sequential which indicates that the substeps should be executed left to right; choice, which allows any one of the substeps to be performed; and try, which indicates that the substeps should be executed in left to right order until one of them succeeds. These latter two are discussed in further detail in Section 3.1.

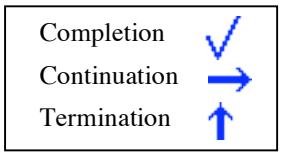


2.2 Exception Handling

Exception handlers are connected via edges attached to the 'X' on the right side of the black bar. Exception handling in Little-JIL is divided into three parts: signaling that an exceptional condition has occurred, determining what steps should be invoked to handle the exceptional condition and then executing them, and finally determining how the process should proceed after the specified steps have been completed.

In a manner similar to statements in traditional programming languages, a Little-JIL step signals an exceptional condition by throwing an exception object. Unlike traditional languages however, Little-JIL steps are guarded by pre- and post-requisites which signal their failure by throwing exceptions as well, functioning much like assert statements. Similar to pre- and post- conditions in some traditional languages, the bundling of a step together with its requisites creates a scope that cleanly separates the task from its checking, but ensures that the step can only be called in the proper context, and specifies the guarantees that the step can make to its callers. As in a traditional language, once an exception has been thrown, Little-JIL determines how the exception should be handled by searching up the call stack. Once the handler has been located and executed, the process specification is consulted to determine how execution should proceed. Unlike traditional languages, which generally only permit the handling scope to complete successfully, or throw an exception, Little-JIL also offers continuation.

Completion, represented iconically by a “check mark” on the edge connecting the handler to its parent step, corresponds to the usual semantics from traditional languages. The associated step is finished, and execution continues as specified by the parent. Continuation, represented iconically by a right arrow, indicates that the associated step should proceed as if the substep that threw the exception had succeeded. It is important to note that this is not resumption – if several levels of scopes had to be searched before finding a matching handler, those scopes have still been exited. Termination, represented by an arrow pointing up, allows the handler to propagate the triggering exception up to an enclosing scope much as in a traditional language.



3 Exception Handling Patterns

In this section we present the exception handling patterns that we have identified in the course of our work in defining processes in different domains. Following in the style introduced in the classic Design Patterns book [2], we present our patterns as a catalog. For each pattern, we provide:

- Its name
- Its intent – what recurring behavior the pattern captures
- Its applicability – in what situations the pattern should be used
- Its structure – the general structure of the pattern expressed in Little-JIL
- An example from the domain of software process that uses the pattern

We organize the patterns into a set of categories. We describe the nature of each category, and then present the specific patterns that it contains.

3.1 Trying Other Alternatives

One common category of patterns describes how to deal with decisions about which of several courses of action to pursue. In some cases, such decisions are based upon conditions that can be encoded directly in the process, essentially using an if-statement to make the choice. In other cases, however, it may be difficult to capture all conditions under which each course of action is best suited. In those cases, it is often most effective to just present alternatives to try. If the alternative that is tried fails, another alternative is to be tried in its place.

In such cases it is often desirable to simply enumerate a set of alternatives without specifying completely the exact conditions under which each alternative is to be taken, but rather using exception handling to move on to untried alternatives. In this category we have identified two different exception handling patterns: unordered alternatives and ordered alternatives.

3.1.1 Pattern Name: Unordered Alternatives

Intent: There may be multiple ways of accomplishing a task. If an exception occurs using one of these ways, an alternative should be tried instead.

Applicability: This pattern is applicable whenever there are multiple ways to accomplish a task and it is not known a priori which is most appropriate. In this case, it is left to the agents to decide which actions to attempt in which order. If an attempted action fails, there is another attempt to complete the task by choosing a different action.

Structure: The Little-JIL diagram shown in Figure 2 indicates the structure of this pattern. Note that, in Little-JIL, decisions that are deferred to agents are represented with a choice step, represented with the  icon. If an exception occurs while executing the chosen step, a *continuation* handler indicates that the agent should be given the opportunity to select another alternative. The structure below shows just

two alternatives, but, in general, there can be an arbitrary number of these. Each time an exception is handled with a continuation handler, that alternative is removed from further consideration.

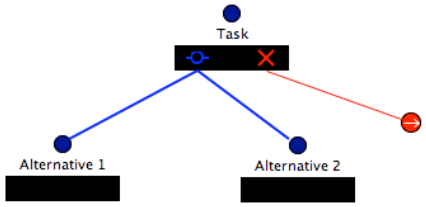


Figure 2: A Little-JIL depiction of the Unordered Alternatives pattern

3.1.2 Pattern Name: Ordered Alternatives

Intent: There are multiple ways to accomplish a task and there is a fixed order in which the alternatives should be tried.

Applicability: This pattern is applicable when there is a preferred order among the alternatives that should be tried in order to execute certain task.

Structure: The Little-JIL diagram in Figure 3 depicts the structure of the Ordered Alternative pattern, which uses Little-JIL's Try step. The children of a try step are alternative ways of completing the task. The alternatives are presented to the agent in order from left to right. If an alternative succeeds, the task is completed and no more alternatives are offered. If execution of an alternative throws an exception that is handled with a *continuation* handler, the next alternative is then presented to the agent.

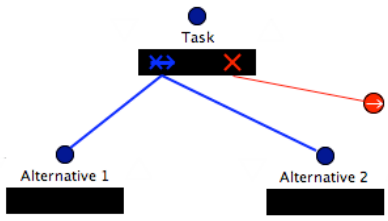


Figure 3: Little-JIL depiction of the structure of the Ordered Alternatives pattern

Sample Code and Usage: The need to select from alternatives arises frequently in most processes. Figure 4 depicts an example of how this may arise in defining the highest level of a process for developing software. In this example, a software company's policy may be to always try to reuse existing code, if possible, in order to reduce development costs. However, if the reuse of existing code modules is impossible under the given circumstances, then it is necessary to do a custom implementation. These alternatives are represented by the use of the Try step semantics in defining the **Implement** step in Figure 4. This example thus shows the

use of the Ordered Alternatives pattern to represent the attempt to employ a reuse approach prior to doing a custom implementation.

There are several possible approaches in trying to reuse existing code. Some examples are employing inheritance and delegation or instantiation of a parameterized class. Knowing which alternative to try first might be left to the judgment of the developers. Figure 4 expresses these alternatives by using Choice step semantics to define the **Reuse existing modules** step. This is an example of the use of the Unordered Alternatives pattern. If the developer's first choice does not work out, this pattern specifies that the agent can then choose one of the remaining alternatives.

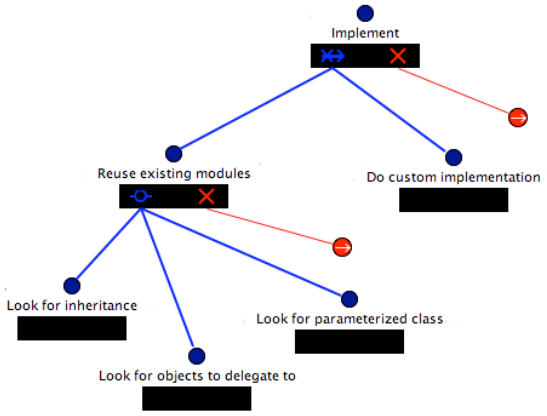


Figure 4: An example of the Ordered Alternatives and Unordered Alternatives patterns.

3.2 Inserting Behavior

Another category of process behaviors that is readily represented by an exception pattern is the case where an exception is used to insert some additional actions that are needed in order to fix problems that have been identified during execution of some activity. We have seen two distinct ways in which such behavior insertion occurs. The first is immediate fixing in which the problems are addressed before continuing with the activity, and the second is deferred fixing, in which the problem is noted and perhaps worked around, and then more fully addressed at some future point. We now specify each as a pattern, and indicate how they can be used to help make the definition of a software development process clear.

3.2.1 Pattern Name: Immediate fixing

Intent: When an exception occurs, some action is taken to fix the problem causing the exception before continuing with the remainder of the process.

Applicability: This pattern allows the insertion of extra behavior to handle expected, but unusual situations. It is useful in situations where an expected error is

likely to occur, where a simple procedure exists to fix the problem, and once fixed, the process can continue as if the error had not occurred.

Structure: Figure 5 shows the Little-JIL structure of this pattern. In this example, an exception is thrown during the execution of **Step 1**. The exception handler **Fix** then takes an action to fix the identified problem and then allows the process to continue with **Step 2**. If no exception occurs, the process goes directly from **Step 1** to **Step 2**.

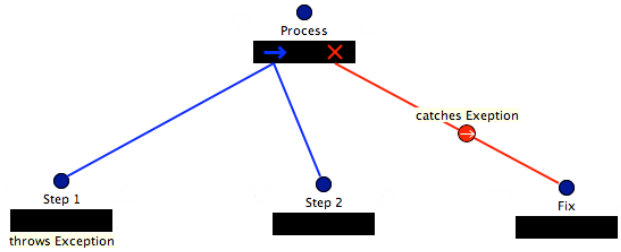


Figure 5: A Little-JIL depiction of the structure of the Immediate Fixing pattern

3.2.2 Pattern Name: Deferred Fixing

Intent: When an exception occurs, action must be taken to record the error and possibly provide partial fixing. Full fixing is either not possible or not necessary immediately. Later in the process, an additional action needs to be executed to complete the recovery from the condition that resulted in throwing the exception.

Applicability: This pattern is useful in allowing the insertion of additional behavior to prevent process execution from coming to a halt. The pattern specifies partial handling of situations that are unusual, yet predictable. This is useful in situations where complete fixing of the exceptional condition is not immediately possible or not desirable (e.g. because it would be too time consuming or disruptive).

Structure: Figure 6 is a Little-JIL depiction of the structure of this pattern. In Figure 6 an exception is thrown during the execution of **Substep 1**. The exception is handled by **Do temporary fix**, an exception handler that would make some expedient temporary adjustment, and then returns to regular processing, as indicated by the continuation handler. However, at some later stage of the process, an additional step (or sequence of steps), represented by the step **Some step**, must be executed to either complete the handling of the exception or check that the appropriate action to handle the exception had been taken at some prior point.

Sample Code and Usage: Figure 7 is the Little-JIL depiction of an example of a high-level software development process that makes use of both the Immediate Fixing pattern and the Deferred Fixing pattern.

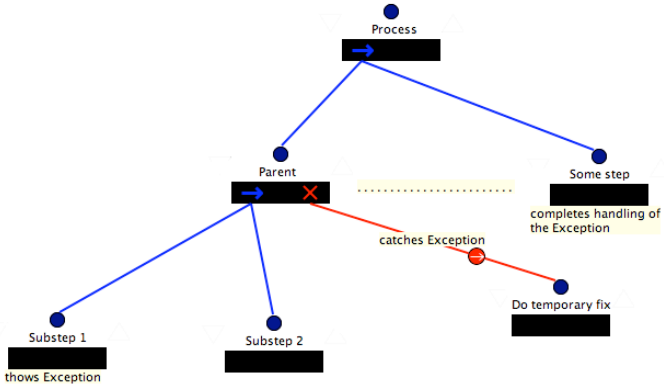


Figure 6: A Little-JIL depiction of the structure of the Deferred Fixing pattern

In this example, if an error occurs during compilation (represented by the postrequisite of the **Write some code** step), the **Fix Compilation Error** exception handler step is then immediately executed and coding continues. This is thus an example of the use of the Immediate Fixing pattern.

In contrast, note that when a failure occurs during execution of the **Run test case** step, the exception handler that is executed, **Log test case failure**, does not fix the bug causing the testcase failure, but rather makes a notation in a test case log so that the bug can be fixed later. After all testing is complete, the test case log is reviewed and the code is fixed by executing the **Fix Bug** step. This is an example of use of the Deferred Fixing pattern.

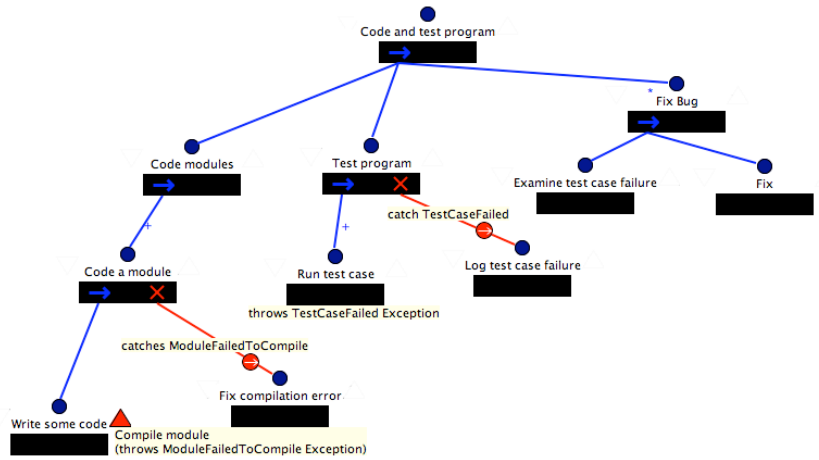


Figure 7: An example of the Immediate Fixing and Deferred Fixing patterns.

3.2.3 Related Pattern: Rework

While the fixes that can be inserted in response to an exception are as varied as the activities and the exceptions themselves, many fixes need to go back and revise the

results of some earlier activity. Cass et. al. [3] argued that rework should be described as re-invocation in a context. This characterization permits us to use the above patterns as context for inserting rework in response to an exception.

Sample Code and Usage: Many phases of software engineering benefit from the Rework pattern. The example process depicted in Figure 8 shows the use of Immediate Fixing to rework a previously created requirement. If during requirements definition, the creation of a subsequent requirements element creates an incompatibility with a requirements element that had been created previously (indicated by an exception thrown during execution of the **Check rqmt OK** postrequisite to step **Declare and define requirement**), it then becomes necessary to rework the previously generated requirements element by reexecuting the step **Develop requirement element**, but now benefiting from knowledge of all of the requirements elements created up to this point (notably the requirements element whose recent creation resulted in the observed incompatibility).

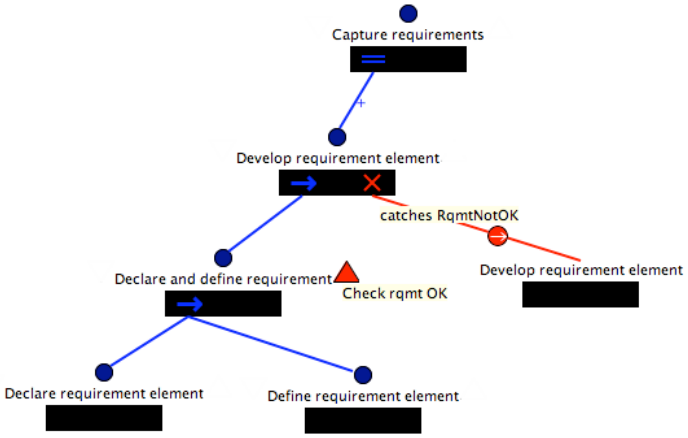


Figure 8: An example of the Rework pattern

3.3 Canceling Behavior

A final category of exception handling patterns is one in which an action being contemplated must not be allowed for some reason.

3.3.1 Pattern Name: Reject

Intent: It sometimes becomes apparent that an action being contemplated should not be allowed. The agent contemplating the action must be notified, and allowed to make adjustments or changes and try again, if so desired.

Applicability: This pattern is often used to create an entry barrier to a part of a process.

Structure: Figure 9 is a Little-JIL depiction of the structure of this pattern. In the process depicted here, the process first gets input from the step **Get process input**, which it intends to use in the step **Use input**. Before using it the process

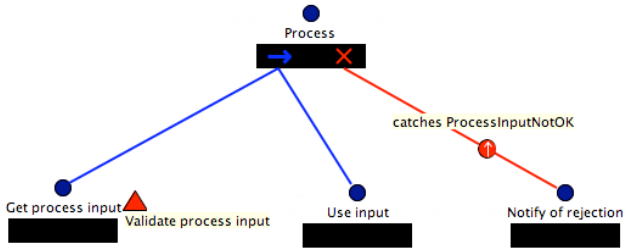


Figure 9: A Little-JIL depiction of the structure of the Reject pattern

validates that the input is acceptable by executing a check in the form of a postrequisite to the **Get process input** step. If the postrequisite fails, an exception is thrown, the exception handler notifies the provider of the input of the problem and then continues forward with the process, effectively disregarding this input. In Little-JIL, validation is typically done as shown here, by attaching a pre- or post-requisite to a step.

Sample Code and Usage: Figure 10 provides an example of the use of this pattern in a software development process. This example shows a process, **Make a good fix**, for fixing a module. The process begins by first checking (in the postrequisite of the **Code improved module** step) to see if we really have improved the module by testing, formal and/or informal analysis. If we decide the purported fix is not really an improvement, we reject the fix, instead of accepting it in the next step.

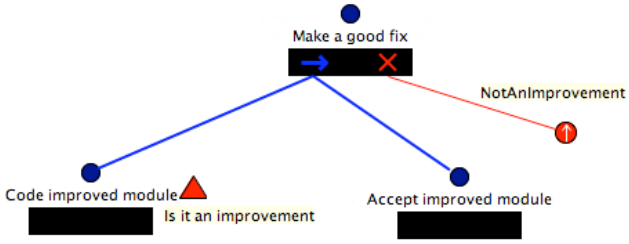


Figure 10: An example of the use of the Reject pattern in software development.

5. Related Work

Exceptional situations commonly arise during the execution of processes. In recognition of this, many process and workflow languages include constructs to allow for the definition of exception handlers (for example, Little-JIL [1], WIDE [4], OPERA [5]). While researchers continue to study how best to define exception handling within processes, exception handling has become more mainstream with its inclusion in process languages like BPEL4WS [6] and products like IBM's WebSphere [7].

Ambler [8], [9] has identified numerous patterns within the domain of software engineering. The patterns that he describes explore approaches to object-oriented software development, capturing the essence of specific software engineering

activities, like technical review or modeling. In contrast, in our work we investigate the role of a specific linguistic construct, namely exception handling, across a range of domains, thereby focusing on general expressiveness of processes.

In more closely related work, Russell, van der Aalst and ter Hofstede [10] have begun to investigate the occurrence of patterns within workflow. They categorize patterns according to the aspect of the workflow definition being emphasized: control flow, data flow, resources and exception handling. They approach exception handling patterns by identifying four dimensions associated with exception handling: the nature of the exception, if and how the work item that encounters the exception continues, whether other work items are cancelled as a result of the exception and whether there is any rollback or compensation performed. Based on this analysis, they consider combinations arising from these four dimensions to derive a universe of possible patterns in a bottom-up fashion.

Our approach differs from the approach of van der Aalst et al. in that it is driven by recognition of patterns that we have seen occur in processes from multiple domains. We thus approach the identification of patterns in a top-down manner, analyzing uses of exception handling to generalize and extract patterns that we believe to be useful beyond the specific processes in which we have found them.

5. Conclusions and Future Work

The exception handling patterns described here are useful in raising the abstraction level of processes. They provide a way of approaching exception handling by providing a framework of questions we can ask. Can we fix the problem immediately? Is there another alternative the process should offer? Should we reject this input entirely?

The examples that we present here show these patterns applied to software engineering examples. We have been actively involved in exploring the use of process technology in other fields such as medical processes and negotiation processes. We are examining the use of exception handling within these processes as well and believe it will affirm our claim that the patterns here are general purpose.

We also continue to examine the expressiveness of Little-JIL. In particular, the combination of choice steps with a continuation handler and try steps with a continuation handler allow for a concise expression of the Ordered Alternatives and Unordered Alternatives patterns, which we believe would be much more difficult with other notations. In contrast, there may be other exception handling patterns that would be easily expressed in languages other than Little-JIL, giving us ideas for how we can improve the language.

6. Acknowlegments

The authors wish to express their gratitude to numerous individuals who have contributed examples and insights supporting the points made in this paper. In particular, we wish to thank Lori A. Clarke, George Avrunin, Beth Henneman, Phil

Henneman, Ethan Katsh, Dan Rainey, Norm Sondheimer, Mohammed S. Raunak, Irina Ros, Rachel Cobleigh, Bin Chen, and Matt Marzilli for conversations, examples, and constructive comments all of which have contributed to this work.

This material is based upon work supported by the US National Science Foundation under Award Nos. CCR-0427071, CCR-0204321 and CCR-0205575. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of The National Science Foundation, or the U.S. Government.

References

1. Wise, A.: Little-JIL 1.5 Language Report. Department of Computer Science, University of Massachusetts, Amherst, MA 01003 (2006)
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Pattenrs: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
3. Cass, A.G., Osterweil, L.J.: Programming Rework in Software Processes. Department of Computer Science, University of Massachusetts, Amherst, MA 01003 (2002)
4. Casati, F., Ceri, S., Paraboschi, S., Pozzi, G.: Specification and Implementaiton of Exceptions in Workflow Management Systems. ACM Transactions on Database Systems (1999)
5. Hagen, C., Alonso, G.: Exception Handling in Workflow Management Systems. IEEE Transaction on Software Engineering **26** (October 2000) 943-958
6. Curbera, F., Khalaf, R., Leymann, F., Weerawarana, S.: Exception Handling in the BPEL4WS Language. Conference on Business Process Management (2003)
7. Fong, P., Brent, J.: Exception Handling in WebSphere Process Server and WebSphere Enterprise Service Bus. http://www.ibm.com/developerworks/websphere/library/techarticles/0705_fong/0705_fong.html. (May 2007)
8. Ambler, S.: Process Patterns: Building Large-Scale Systems Using Object Technology. Cambridge University Press (1998)
9. Ambler, S.: More Process Patterns: Delivering Large-Scale Systems using Object Technology. Cambridge University Press (1999)
10. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Exception Handling Patterns in Process-Aware Information Systems. BPM Center Report (2006)